2007

Architectural Design, Behavior Modeling and Run-Time Verification of Network Embedded Systems

Shing, Man-Tak

Architectural Design, Behavior Modeling and Run-Time Verification of Network Embedded Systems

Man-Tak Shing[1] and Doron Drusinsky[1,2]

[1] Department of Computer Science, Naval Postgraduate School
Monterey, CA 93943-5118, USA
{shing, ddrusin}@nps.edu
[2] Time Rover Inc.
Cupertino, CA 95014, USA
dorond@time-rover.com
http://www.time-rover.com

Abstract. There is an increasing need for today's autonomous systems to collaborate in real-time over wireless networks. These systems need to interact closely with other autonomous systems and function under tight timing and control constraints. This paper concerns with the modeling and quality assurance of the timing behavior of such network embedded systems. It builds upon our previous work on run-time model checking of temporal correctness properties and automatic white-box testing using run-time assertion checking. This paper presents an architecture for the network embedded systems, a lightweight formal method that is based on formal statechart assertions for the design and development of networked embedded systems, and a process of using run-time monitoring and verification, in tandem with modeling and simulation, to study the timing requirements of complex systems early in the design process.

Keywords: Network Embedded System, Lightweight Formal Method, Architecture Design, Run-Time Verification, Statechart Assertions.

1 Introduction

With the recent advance in Internet and wireless technology, there is a new demand for high performance and intelligent automobiles, aircraft and autonomous robots to collaborate in real-time over wireless networks. These systems need to interact closely with other embedded systems and function under tight timing and control constraints. This paper addresses the need to verify the timing properties of real-time, reactive distributed systems. It presents a testing methodology that builds upon our work on run-time model checking of temporal correctness properties and automatic white-box testing using run-time assertion checking [4]. Run-time Execution Monitoring of formal specification assertions (REM) is a class of methods for tracking the temporal behavior, often in the

F. Kordon and J. Sztipanovits (Eds.): Monterey Workshop 2005, LNCS 4322, pp. 281–303, 2007.

form of formal specification assertions, of an underlying application. REM methods range from simple print-statement logging methods to run-time tracking of complex formal requirements (e.g., written in temporal logic or as statechart assertions) for verification purposes. NASA used REM for the verification of flight code for the Deep Impact project [10]. In [8], we showed that the use of run-time monitoring and verification of temporal assertions, in tandem with rapid prototyping, helps debug the requirements and identify errors earlier in the design process. Recently, REM has been adopted by the U.S. Ballistic Missile Defense System project as the primary verification method for the new BMDS battle manager because of its ability to scale, and its support for temporal assertions that include real-time and time series constraints [2].

The rest of the paper is organized as follows. Section 2 provides an overview of the StateRover statechart assertion formalism. Section 3 presents an architecture that supports high-level specification of network level objectives and policies and the enforcement of these policies by direct re-configuration of the states of individual network elements. We will illustrate the proposed architecture with an example from the automatic highway platoon system. Section 4 describes the use of run-time monitoring and verification, in tandem with modeling and simulation, to study the timing requirements of complex systems early in the design process. Section 5 presents a discussion on the approach and Section 6 draws some conclusions.

2 The Statechart Assertions

Harel Statecharts [15] are commonly used in the design analysis phase of an object oriented UML based design methodology to specify the dynamic behavior of complex reactive systems. In [5] [6], Drusinsky presented a new formalism that combines UML-based prototyping, UML-based formal specifications, run-time monitoring, and execution-based model checking. The new formalism is supported by StateRover, a commercially available tool from the Time Rover Inc. StateRover provides support for design entry, code generation, and visual debug animation for UML statecharts combined with flowcharts. The new formalism and tool allow system designers to embed deterministic and non-deterministic statechart assertions in statechart designs and execute the assertions in tandem with their primary UML statechart to provide run-time monitoring and run-time recovery from assertion failures.

2.1 A Statechart Example

Figure 1 shows the top-level statechart of a leader election (*LE*) module, which is one of the many leader election modules connected by a unidirectional ring network. The top-level statechart consists of three states, the Initializing state and two composite states named *Electing_Leader* and *Found_Leader*, together with a set of state variables declared in the associated local variable declaration box shown in Figure 1. Each *LE* module uses the *Own_Id* variable to store its unique integer identity and uses the *Leader_Id* variable to remember the identity

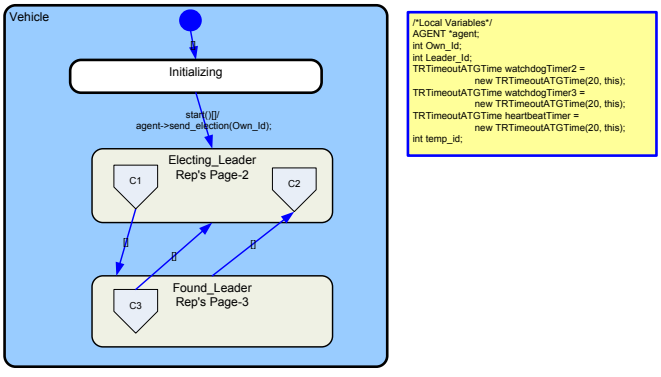


Fig. 1. Top-level page of the *LE* statechart

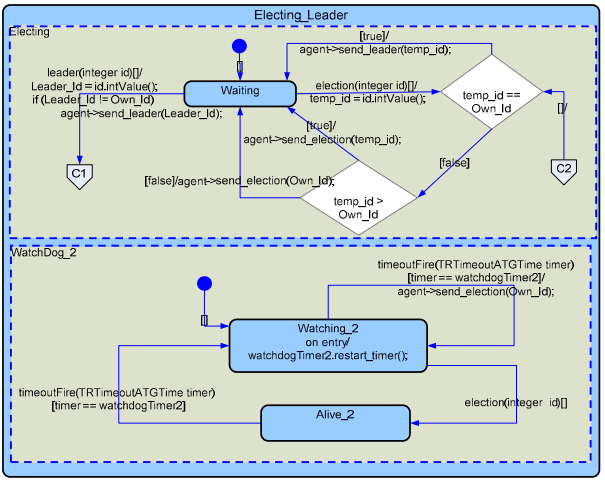


Fig. 2. The Electing_Leader statechart of the *LE* module

of the current leader, which is the largest identity value among the identities of all the active *LE* modules in the network. In addition, it has three timers and an agent object. The timers are instances of a built-in StateRover timer class, and the agent object serves as a proxy for all network communications.

The *LE* statechart starts at the Initializing state waiting for the arrival of the *start()* event from the environment. Upon receiving the *start()* event, it uses the *send_election()* method of its agent object to send an *election()* message to its neighbor in the network and then enters the *Electing_Leader* composite state shown in Figure 2. The *Electing_Leader* composite state consists of two concurrent threads, *Electing* and *Watchdog_2*. The statechart in the *Electing* thread models the logic of the following simple leader election algorithm.

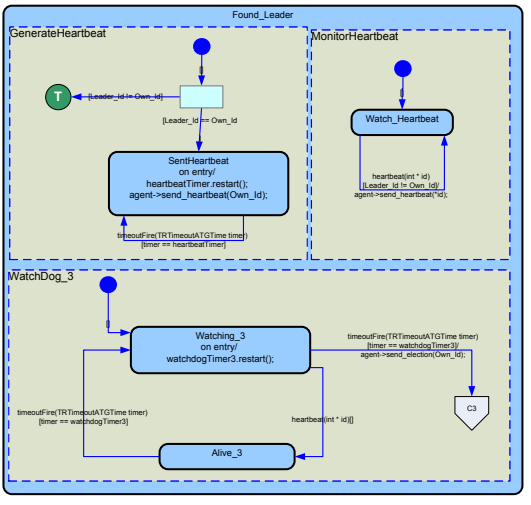


Fig. 3. The Found_Leader statechart of the *LE* module

```
if (event == election(id)) then // on-going election
{
  if (id == Own_Id) then
    send leader(Own_Id} event to its neighbor;
  else if (id < Own_Id) then
    send election(Own_Id) event to its neighbor;
  else
    send election(id) event to its neighbor;
}
else if (event == leader(id)) then
{ // found leader, terminate election
  Leader_Id = id;
  if (Leader_Id != Own_Id) then
    send leader(id) event to its neighbor;
  reset the watchdogTimer_2 timer;
  transition to the Found_Leader state via the page connector C1;
}
```

The statechart in the *Watchdog_2* thread makes sure that the *LE* statechart receives at least one *election()* message every 60-second cycle while it is participating in an on-going election. If the *LE* statechart receives the event *timeoutFire(timer)* with *timer == watchdog2Timer* while it is in the *Watching_2* state, it will initiate another round of leader election by sending an *election(Own_Id)* message to its neighbor because it has not received any *election()* message within the last cycle.

The leader election algorithm terminates when the *LE* statechart receives a *leader(id)* message, and will transition from the *Electing_Leader* state to the

Found_Leader state via the page connector $C1$. Note that the correctness of the algorithm relies on a reliable and fully trusted network of cooperating *LE* modules.

Figure 3 shows the statechart of the *Found_Leader* composite state. It consists of three concurrent threads. While in the *Found_Leader* state, the leader uses the statechart in the *GenerateHeartBeat* thread to send out a *heartbeat(Own_Id)* message once every 60 seconds, and each *LE* statechart expects to receive at least one *heartbeat()* message from the leader via its neighbor in every 60-second cycle. If the *LE* statechart receives the event *timeoutFire(timer)* with *timer == watchdog3Timer* while it is in the *Watching_3* state, it will initiate another round of leader election by sending an *election(Own_Id)* message to its neighbor and transitioning to the *Electing_Leader* state because it has not received any *heartbeat()* message within the last cycle. Other *LE* modules will also enter the *Electing_Leader* state via the page connector $C2$ when they receive the *election()* messages from their neighbors in the network while they are in the *Found_Leader* state.

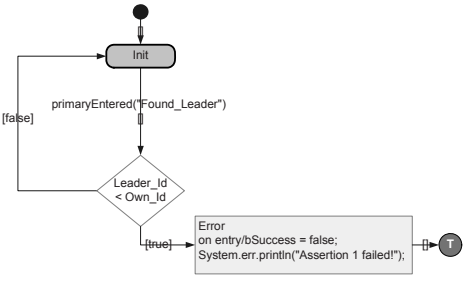
2.2 Statechart Assertions

Studies have suggested that the process of specifying requirements formally enables developers to gain a deeper understanding of the system being specified, and to uncover requirements flaws, inconsistencies, ambiguities and incompletenesses [11]. The StateRover uses deterministic and non-deterministic statecharts for the formal specification of temporal correctness properties (i.e. properties about the correct ordering, sequencing, and timing of events and responses). Figure 4 contains three statechart assertions for the following natural language requirements:

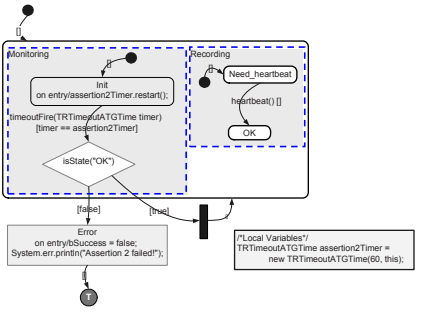
Assertion 1. Leader_Id must be greater than or equal to Own_Id whenever the LE statechart enters the Found_Leader state.
Assertion 2. At least one heartbeat occurs every 60 seconds while LE is in the Found_Leader state.
Assertion 3. There should not be 3 or more rounds of leader election within a 5-minute interval.

Assertion 1 is an example of a correctness-property assertion that ensures "the leader election algorithm correctly selects the active *LE* module with the largest identity value as the leader". Assertion 2 is an example of a timing constraint assertion that ensures "the *LE* module receives at least one heartbeat every 60 seconds while it is in the *Found_Leader* state". Assertion 3 is an example of a temporal constraint assertion to ensure the stability of the networked system.

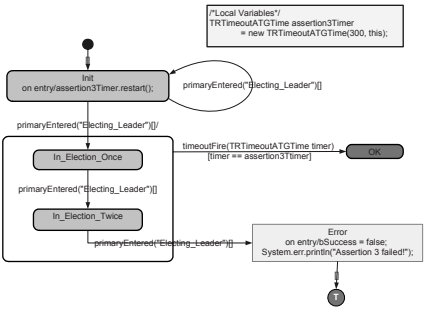
Figures 5-6 show the combined *LE* statecharts with embedded statechart assertions, where the Assertion1 and Assertion3 statecharts now become substatecharts of the top-level *LE* statechart, and the Assertion2 statechart becomes a sub-statechart of the *Found_Leader* state. Sub-statecharts represent whole statecharts defined elsewhere, i.e., in a different statechart file. Using sub-statecharts facilitates reuse: the assertion statecharts are drawn once but can be reused many times in many other statecharts. A statechart with an embedded substatechart is called a *primary* statechart. StateRover provides a way to map the events



(a) Assertion1 Statechart



(b) Assertion2 Statechart



(c) Assertion3 Statechart

Fig. 4. Assertion statecharts

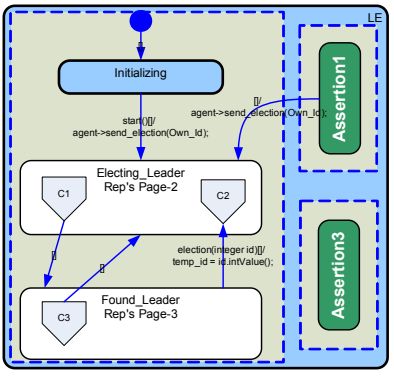


Fig. 5. The top-level page of the *LE* statechart with embedded assertion sub-statecharts

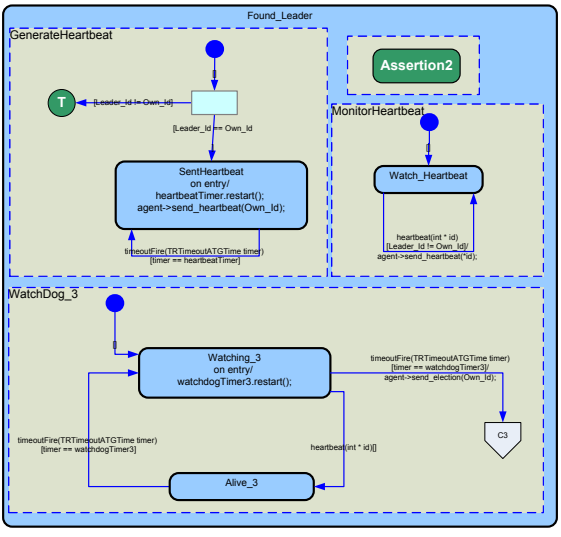


Fig. 6. The Found_Leader statechart with embedded Assertion2

between the primary statechart and its sub-statecharts. The StateRover's code generator generates code that automatically passes down events from the primary statechart to the sub-statecharts.

Figures 4a and 4b are examples of deterministic statechart assertion. (Non-deterministic assertions are discussed in Section 2.3.) Every time the *Found_Leader* state of the primary (i.e., *LE*) statechart is entered, the *Assertion2* state is entered, and the sub-statechart becomes active, starting its computation in the initial states (*Init* and *Need_heartbeat* in Figure 4b). The sub-statechart remains active until either assertion2Timer fires its *timeout* event and the *Recording* thread is not in the *OK* state, or the *LE* statechart exits the *Found_Leader* state. Once the *LE* statechart leaves the *Found_Leader* state, it is no longer in the *Assertion2* state and the assertion sub-statechart will not be executing at all. The next time the *LE* statechart enters the *Assertion2* state, the assertion sub-statechart starts its computation from its initial states all over again.

The statecharts in Figure 4 are formal specifications that assert about the primary *LE* statechart because they each make a statement about the correctness of the primary statechart. They do so using a built-in Boolean variable named *bSuccess*, and a corresponding method called *isSuccess*(), both auto-generated by the StateRover's code generator. Whenever the assertion detects a violation of the requirement it sets *bSuccess* = false, as in Figure 4b when assertion2Timer fires its *timeout* event and the *Recording* thread is not in the *OK* state. The method *isSuccess()* returns the value of *bSuccess* mainly for the purpose of JUnit testing and automatic white-box testing described in Section 2.5 and Section 2.6. Because statechart assertions are usually used to flag errors, the *isSuccess()*, which monitors the *bSuccess* variable, is set to true by default. It is the assertion

developers' responsibility to set it to false when the assertion fails, as done in the Error activity box in Figure 4b.

In addition, an unlabeled transition from the *Assertion1* state to the *Electing_Leader* state is added to enable run-time recovery (Figure 5). Whenever the Assertion 1 fails, because *Leader_Id* < *Own_Id*, the sub-statechart reaches the terminal state (T) and will therefore cause the unlabeled transition out of the *Assertion1* state to fire, forcing the *LE* statechart to transition to the *Electing_Leader* state. Consequently, the *LE* statechart recovers from the specification failure by starting another round of leader election.

2.3 Non-deterministic Assertion Statecharts

The StateRover supports the specification of more complex requirements using non-deterministic statecharts. Figures 4c is an example of non-deterministic statechart assertions. While deterministic statechart assertions suffice for the specification of many requirements, theoretical results [7] show that non-deterministic statecharts are exponentially more succinct than deterministic Harel Statecharts. As indicated in Figure 4c, there is an apparent next-state conflict when event *primaryEntered("Electing_Leader")* is sensed while the statechart is in the *Init* state. The vanilla StateRover code generator (described in the next section) generates an error message for such a statechart. It is however a legal *non-deterministic* statechart. Non-deterministic statecharts use a special StateRover code generator that creates a plurality of *state-configuration* objects, one per possible computation in the assertion statechart. Non-deterministic statechart assertions use an *existential* definition of the *isSuccess()* method, where if there exists at least one state-configuration that detects an error (assigns *bSuccess*=false) then *isSuccess()* for the entire non-deterministic assertion returns false. Likewise, terminal state behavior is existential; if at least one state-configuration is in a terminal state then the non-deterministic statechart assertion wrapper considers itself to be in a terminal state. The StateRover also has a power-user priority mechanism to change or limit the existential default definitions of *isSuccess()* and the terminal state. This mechanism is described in details in [6].

2.4 The StateRover Code Generator

The primary StateRover rapid prototyping tool is its code generator. The StateRover's code generator generates a class per statechart model (i.e. per statechart file) in either Java or C++ language, a convenient level of encapsulation for a controller statechart that lives within a heterogeneous system of Java or C++ objects created by various tools or perhaps hand-coded. The class can then be dynamically instantiated according to the needs of the system.

In our example, we have four statechart diagram files, with the *LE* statechart in the first file and the *Assertion1, Assertion2* and *Assertion3* sub-statecharts in the second, third and the fourth files. The StateRover's code generator automatically connects the four statecharts objects resulting in an executable *LE*

module. The controller class consists of a set of event handlers (one per transition event), the central event dispatcher *execTReventDispatcher*, and the source code for local variable declarations and methods supplied by the users via the dialog boxes of StateRover's statechart editor. In addition, the code generator also generates a Java interface, named LEIF, to allow the test drivers or other systems from the external environment to interact with the *LE* module.

Statechart orthogonality is implemented by the vanilla code generator using a fixed schedule created during code generation. For example, in Figure 3, three orthogonal *timeoutFire()* transitions, two in the *WatchDog_3* thread and one in the *GenerateHeartbeat* thread, will be realized as three if blocks within the *timeoutFire()* event handler. The order of these if blocks induces a fixed firing schedule for corresponding transitions. Besides the vanilla code generator, the StateRover has a concurrent code generator that generates multi-threaded Java code for statecharts with Harel-concurrence.

2.5 Testing of Generated Code

The generated code is designed to work with the JUnit Test Framework [1] [18]. Use Case scenarios used by the system designers to identify user needs and system requirements are hand-coded as JUnit test cases and exercised against the generated statechart code. Figure 7 illustrates the StateRover's JUnit based testing architecture. Tests, which consist of sequences of events and timing information, are either hand coded or auto-generated by the white-box test generator.

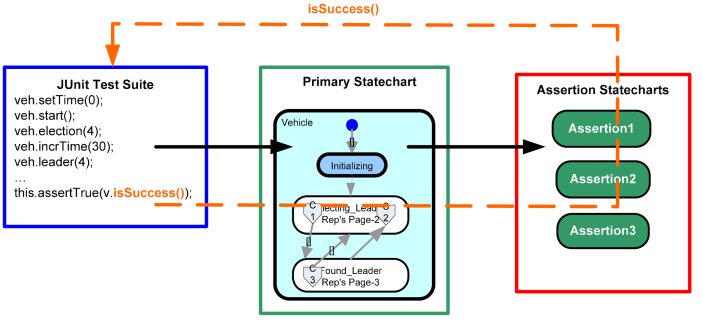


Fig. 7. JUnit based simulation and testing architecture

For example, the following hand-code test case describes a scenario in which the *LE* statechart successfully participates in two rounds of leader election within an interval of 180 seconds.

```
import junit.framework.*;
public class TestLE1 extends TestCase {
  private LE veh = null;
  private AGENT t = null;
```

```
  public TestLE1(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    t = new AGENT();
    veh = new LE(3, -1, t); // Own_Id == 3,
                            // Leader_Id == -1
  }

  protected void tearDown() throws Exception {
    veh = null;
    super.tearDown();
  }

  // Test Scenario:
  public void testExecTReventDiapatcher() {
    // first round of leader election
    veh.start();
    veh.election(4);
    veh.incrTime(30); // advance clock by 30 sec
    veh.leader(4); // found leader with id == 4

    // veh should now be in the Found_Leader state
    this.assertTrue(veh.isState("Found_Leader"));
    veh.incrTime(120); // advance clock by 120 sec

    // veh should initiate the second round of leader
    // election since it has not received any
    // heartbeat() for more than 60 sec
    this.assertTrue(veh.isState("Electing_Leader"));
    veh.election(1);
    veh.incrTime(15); // advance clock by 15 sec
    veh.election(3);
    veh.incrTime(15); // advance clock by 15 sec
    veh.leader(3); // found leader with id == 3

    // veh should be in the Found_Leader state
    this.assertTrue(veh.isState("Found_Leader"));

    // the testcase should return bSuccess == true
    this.assertTrue(veh.isSuccess());
  }
}
```

A test exercises the primary statechart model, which then automatically exercises embedded assertions. The assertion feeds back a Boolean success value, *isSuccess()*, to the JUnit based test, which then announces fail or success accordingly.

It is important to validate the correctness of the assertions early in the software development process. By keeping each statechart assertion in a separate diagram file, we can generate code and test each statechart assertion independent of the prototype design. For example, a developer might expect the following scenario to cause the Assertion2 statechart to fail, since the heartbeats do not arrive regularly; the inter-arrival time between the second and the third heartbeat is more than 60 seconds.

```
public class TestAssertion2 extends TestCase {
  private Assertion2 assert2 = null;
  ...
  protected void setUp() throws Exception {
    super.setUp();
    assert2 = new Assertion2();
  }
  ...
  // Test Scenario:
  public void testExecTReventDiapatcher() {
    assert2.heartbeat(4); // receive first heartbeat
    assert2.incrTime(61); // advance clock by 61 sec
    assert2.heartbeat(4); // receive second heartbeat
    assert2.incrTime(117); // advance clock by 117 sec
    assert2.heartbeat(4); // receive third heartbeat
    assert2.incrTime(10); // advance clock by 10 sec
    assert2.heartbeat(4); // receive third heartbeat
    // the testcase should return bSuccess == false
    this.assertFalse(assert2.isSuccess());
  }
}
```

The developer of the assertion was surprised by the assertion's success for this scenario. After a closer examination of the natural language assertion and the Assertion2 statechart, the developer decided that the error was caused by an incorrect interpretation of the natural language requirement "at least one heartbeat occurs every 60 seconds". Hence, he reformulated the natural language requirement and the Assertion2 statechart as follows:

Assertion 2 (revised). The inter-arrival time between two consecutive heartbeats cannot exceed 60 seconds.

When the extended primary statechart of Figure 6 (with the revised Assertion2 of Figure 8) is executed using a scenario similar to TestAssertion2, the revised Assertion2 statechart will detect an error in the primary statechart as it is performing REM of the primary. Automatic test generation discussed in Section 2.6

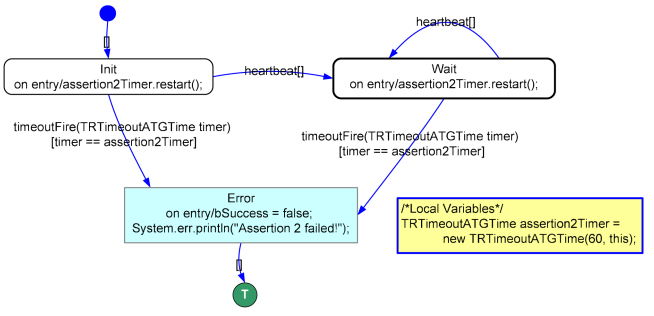


Fig. 8. Revised Assertion2 statechart

is a technique for automatically discovering violating scenarios such as TestAssertion2. This example highlights the subtleties in creating correct formal assertions and the value of testing executable formal assertions via JUnit-based simulations.

2.6 Automatic, Intelligent, White-Box Test Generation

The StateRover's White-Box Test Generator (WBTG) is an automatic code generator for a JUnit TestCase class. This TestCase, however, does not capture a single, human created, scenario as JUnit TestCase objects usually do. Rather, it contains a loop that creates a plurality of tests for a Statechart Under Test (SUT). We denote this WBTG generated TestCase the *WBTestCase*. The auto-generated WBTestCase is usable verbatim within a broader test suite that replaces the one shown in Figure 7, which may include both WBTestCase and the manually created tests. Automatically generated tests are used in three ways:

1. To search for severe programming errors, of the kind that induces a JUnit error status, such as *NullPointerException*.
2. To identify tests that violate temporal assertions. Such failed assertion are captured by a JUnit *assertFalse*() (versus the *assertTrue()*) statement using the *isSuccess*() feedback loop depicted in Figure 7.[1]
3. To identify input sequences that lead the SUT to particular states of interest.

The StateRover generated *WBTestCase* creates sequences of events and conditions for the SUT. The WBTestCase is intelligent in the following regard: it creates only sequences which "matter" to either the SUT or to some assertion statechart. For example, in Figure 1, upon startup, the SUT has one option only: to observe the *start* event. The WBTestCase therefore generates this event. Consequently, the SUT moves to the *Electing_Leader* state shown in Figure 2. There, the SUT expects a *timeout* event, an *election* event, or a *leader* event; so the WBTestCase generates one of those using one of the two algorithms the user

[1] To help statechart designers pinpoint specific errors, each failed test run is reported with an identification number. The causes of failure for a specific run can be investigated in detail by running the automatic white box tester in single test/run mode. Such mechanism helps developers to efficiently eliminate errors in their design.

selects: the *stochastic* algorithm or the *deterministic* algorithm described below. Hence, in general, the StateRover generated WBTestCase repeatedly observes all events that potentially affect the SUT when it is in a given state configuration, selects one of those events and fires the SUT using this event. The WBTestCase auto-generates three artifacts:

1. Events, as described above.
2. Time advance increments, for the correct generation of timeoutFire events.
3. External data objects of the type that the statechart prototype refers to.

This process describes the model-based aspect of the StateRovers WBTG. However, the StateRovers WBTG actually observes all entities, namely, the SUT and all embedded assertions. It collects all possible events from all those entities, thus creating a hybrid model-based and specification-based WBTG. For a SUT with a loop, there is an infinite number of input sequences of unbounded lengths. The StateRovers WBTG addresses these issues in the following manner:

1. The StateRover's user specifies the maximal number of test sequences the WBTestCase is allowed to generate, denoted as the *WB test-budget*.
2. The StateRover's user specifies the maximal length of any test sequence generated by the WBTestCase.

The WBTG uses two primary methods, a stochastic method and a deterministic method, for test generation. For each of three artifacts of concern, namely, the set of possible events, the set of objects the object factory can generate, and simulation time increments, the stochastic method rolls the dice and makes a selection accordingly, while the deterministic method attempts to systematically cover all possible sequences by enumerating these artifacts and traversing new sequences one by one. In addition, the StateRovers WBTG can also be configured to use NASAs Java Pathfinder (JPF) [16]. JPF uses a customized Java Virtual Machine to detect the presence of concurrency errors such as deadlock under varying firing schedules of concurrent transitions and actions. Moreover, JPF can be viewed as a sophisticated hybrid of the deterministic and stochastic methods. JPF makes sure to not revisit system states more than once by recording the state space being visited. The drawbacks of using JPF are: (1) JPF tends to run out of memory for complex systems, (2) JPF wastes resources by model-checking the assertions and the methods of the actions and activities, and (3) JPF does not work well with frameworks like JUnit and Spring.

3 Adaptive Network Architecture

Any good architecture for the networked embedded systems must be adaptive and easily reconfigurable at runtime to cope with the changing environment and to accommodate changing mission needs. In particular, we must move away from today's box-centric network architecture where the decision-making logic of the control plane is spread out over the routers and switches in support of the simple best effort distributed protocol. It is very difficult, if not impossible, to

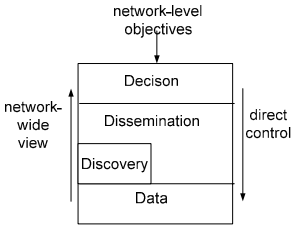


Fig. 9. The 4D architecture (after [13])

retrofit this box-centric architecture to support the more sophisticated network objectives like traffic engineering, survivability, security, and policy management, which require the increasing run-time use of reflection (systems that utilize their own models), self-adaptation, and self-optimization to satisfy the mission needs.

In [13], Greenberg et al proposed a novel clean slate 4D architecture that abstracts the network management into four components – the data, discovery, dissemination and decision planes (Figure 9). The decision plane replaces today's management plane. It consists of multiple servers (called decision elements) that make all decisions driving the network-wide control based on the real-time network-wide view coming from the discovery plane. The discovery plane is responsible for maintaining and updating information about the physical entities making up the network and synthesizes the information into a network-wide view for the decision elements. The data plane handles individual packets based on the output (e.g. forwarding table, packet filters, link scheduling weights, etc.) from the decision plane. The data plane may also collect data on behalf of the discovery plane. The dissemination plane decouples the decision plane from the discovery and data planes. It provides the option to move management information from the decision plane to the data plane and state information from the discovery plane to the decision plane using separate communication paths to avoid the need to establish routing protocols before they can be used.

3.1 Network Architecture for the Automatic Highway Platoon System

In this section, we will illustrate the 4D architecture with an example from the automated highway system, where each vehicle is equipped with vehicle-to-vehicle and vehicle-to-roadside Intelligent Transportation System (ITS) wireless communication, radars for measuring the inter-vehicle distances, sensors for measuring the vehicle's lateral position relative to the lane center, electronically controlled steering, throttle and brake actuators, and the computers for processing data from the sensors and generating commands to the actuators. In addition, the roadway is instrumented with magnetic markers buried along the centerline of each lane at four feet spacing. The magnetic markers enable the vehicle to detect its lateral position, and by alternating the polarities of the magnetic markers, they can also transmit roadway characteristics such as upcoming

road geometry information, milepost, entrances and exits information to the vehicle [14]. The main objectives of the automated highway system are to reduce traffic congestions, enhance safety, and reduce human stress. These goals can be achieved using the Adapted Cruise Control technology [27], which uses sensors to maintain a small but safe inter-vehicle distance between cars while they traveling at high speed in the formation of a platoon [26]. These platoons are coordinated using a leader-follower architecture that centralizes the coordination on the leader [25].

In a centralized platoon, the task of communication to coordinate the vehicle platoon formation is only executed by the leader vehicle. To maintain the platoon formation, the leader (head vehicle) is the only entity that can give order (e.g. velocity, inter-vehicle distance, time and location for lane change, etc.) to its followers, while the followers can only apply the requested changes as well as submit requests for leaving the platoon to the leader. The leader vehicle also has to communicate with other platoons (which can be defaulted to a single vehicle) to coordinate platoon merging as well as safe lane changes, and to communicate with the roadside ITS for real-time traffic information and rules of engagements. Finally, if the leader itself has to leave the platoon, then it must issue order for the followers to select a new leader and hand over the command to the new leader before leaving the platoon.

To support the complex communication requirements of the platoon leader, we propose to apply the 4D architecture to the design of the automated highway system communication network. Figure 10 shows the UML-RT model of the high-level architecture of a vehicle for the automated highway system. UML-RT [24] is an extension of the original UML based on the concepts in the ROOM language [23], and forms the basis for the new features in UML 2.0 for modeling large-scale software systems.

The UML-RT model shown in Figure 10 consists of a set of Vehicle capsules. Each Vehicle capsule consists of a Planning capsule, a Coordination capsule, a Vehicle Control capsule, a Comms capsule and a set of sensor capsules. The Comms capsule, which represents the wireless communication subsystem of the vehicle, is made up of a Decision Element capsule, a Dissemination Element capsule, a Discovery Element capsule and one or more Data Element capsules (as indicated by the multi-object icon). Each Decision Element capsule has two ports to communicate with the Planning capsule and the Dissemination Element module. Each Dissemination Element capsule has multiple ports for communication with its associated Data Element capsules and uses single ports to communicate with the Decision Element capsule and the Discovery Element capsule, as well as forwarding management information from the Decision Element capsule of the platoon leader to the Data Element capsule of the followers via the wireless network in the Environment capsule.

Each vehicle's Decision Element can be in one of the following 3 states – *active*, *voting* or *inactive*. The Decision Elements will be in the voting state when they are in the process of electing a platoon leader. Once selected, the decision element of the platoon leader becomes active and is responsible for notifying the

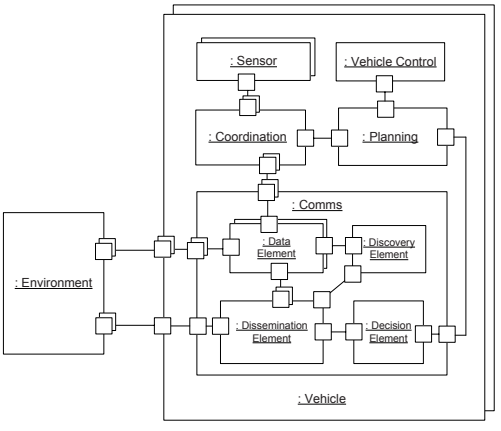


Fig. 10. The high-level architecture of a platoon vehicle

rest of the platoon about the election result. It will create the routing tables for inter-platoon, intra-platoon, as well as vehicle-to-roadside communications, and disseminate the intra-platoon routing table to the followers. The Decision Elements of the other platoon members will become inactive and turn over the control of its Dissemination, Discovery and Data Elements to the newly elected leader.

With the help of the real-time network-view from the discovery plane, the active Decision Element can quickly adapt to changes in the network environment while still maintaining the overall network objective. For example, when communication degrades, the Decision Element can make decision, in real-time, as to which messages should be sent first (via which data paths), which messages should be delayed, or even not be sent at all.

4 Modeling, Simulation and Run-Time Verification

The analysis and design of complex safety-critical networked embedded systems pose many challenges. Feasible timing and safety requirements for these systems are difficult to formulate, understand, and meet without extensive prototyping. Traditional timing analysis techniques are not effective in evaluating time-series temporal behaviors (e.g. the maximum duration between consecutive missed deadlines must be greater than 5 seconds). This kind of requirements can only be evaluated through execution of the real-time systems or their prototypes. Modeling and simulation holds the key to the rapid construction and evaluation of prototypes early in the development process.

4.1 The OMNeT++ Model

OMNeT++, which stands for *Objective Modular Network Testbed in C++*, is an object-oriented discrete event simulator primarily designed for the simulation

of communication protocols, communication networks and traffic models, and multi-processors and distributed systems models [20].

Figure 11 shows a simple OMNeT++ model of a platoon system with three vehicles communicating with one another via the 4D architecture. OMNeT++ provides three principal constructs (*modules*, *gates* and *connections*) for modeling the structures of a target system. An OMNeT++ simulation model consists of a set of modules communicating with each other via the sending and receiving of messages. Modules can be nested hierarchically. The atomic modules are called simple modules; their code are written in C++ and executed as coroutines on top of the OMNeT++ simulation kernel. *Gates* are the input and output interfaces of the modules. Messages are sent out through output gates of the sending module and arrive through input gates of the receiving module. Input and output gates are linked together via connections. *Connections* represent the communication channels and can be assigned properties such as propagation delay, bit error rate and data rate. Message can contain arbitrarily complex data structures and can be sent either directly to their destination via a connection or through a series of connections (called route).

4.2 Integrating StateRover Startchart Designs with OMNeT++ Models

Figure 12a shows an object model of the Decision Element Capsule package, which is made up of a Decision Element class and the statechart classes generated from one or more statechart files. Figure 12b shows an instance of the Decision

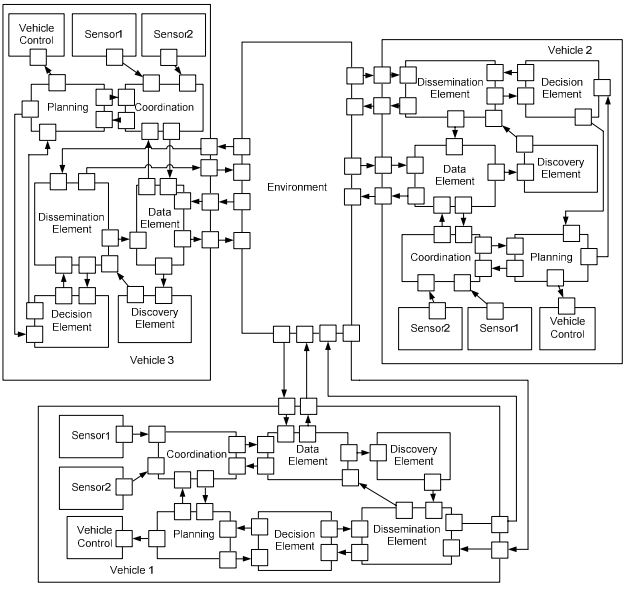


Fig. 11. The OMNeT++ model of an automated highway system

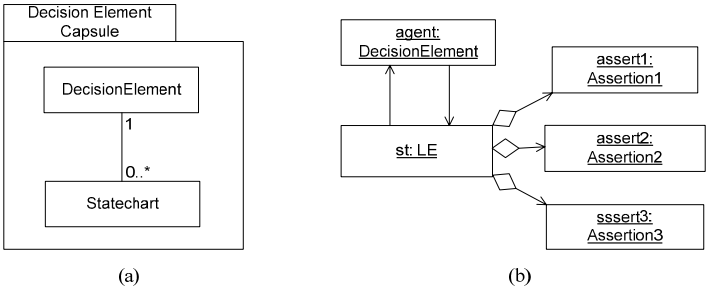


Fig. 12. The object model of the Decision Element Capsule package

Element Capsule package, which contains an instance of the Decision Element class together with an instance of the *LE* statechart for leader election.

Here, the Decision Element class extends the OMNeT++ cSimpleModule class and serves as the proxy (in place of the AGENT class) to handle as all network communications for the *statechart* object.

OMNeT++ uses messages to represent events. Each event is represented by an instance of the cMessage class or one its subclasses. Messages are sent from one module to another – this means that the place where the "event will occur" is the message's destination module, and the model time when the event occurs is the arrival time of the message. Events like "timeout expired" are implemented by the module sending a message to itself. OMNeT++ cSimpleModule class provides a virtual member function, handleMessage(), which will be called for every message that arrives at the module. In our example shown in Figure 12b, we will insert the following code into the handleMessage function of the Decision Element module to handle incoming messages to the module:

```
void DecisionElement::handleMessage(cMessage *msg) {
  if (msg->hasBitError()) {
  // log error and update error count
  }
  else if (msg->isSelfMessage()) // it is a timeout message
    processTimer(msg);
  else {
    switch (msg->kind()) {
      case ELECTION_MSG:
        // extract data from the message and invoke the
        // election() function of the st object
        st->election(msg->getValue());
        break;
      case LEADER_MSG:
        // extract data from the message and invoke the
        // leader() function of the st object
        st->leader(msg->getValue());
        break;
```

```
      case HEARTBEAT_MSG:
        // extract data from the message and invoke the
        // heartbeat() function of the st object
        st->heartbeat(msg->getValue());
        break;
      default: // unrecognized message kind
        // log error and output error message
        break;
    }
  }
}
```

In addition, the Decision Element module provides three functions, *send_ election(int id)*, *send_leader(int id)* and *send_heartbeat(int id)* for the *LE* module to send messages to the other *LE* modules in the network. When invoked by the *LE* object, these functions will create an instance of the cMessage class with an integer value set to *id* and the messageKind set to the appropriate kind, and then send the message via the "to_dissemination" output gate of the Decision Element module.

5 Discussions

5.1 Related Works

With the popularity of UML, Harel statecharts have become the tool of choice for most engineers to design complex reactive systems. While statecharts can effectively specify what a system *should do* (positive information), they tend to be less effective for the specification of safety requirements (i.e., negative information about what a system *must not do*). Hence, researchers have attempted to augment statechart specifications with other formalisms like process algebra [21], symbolic timing diagrams [19] and temporal logic [12], and demonstrated formal proofs for certain properties of the Statechart design. In the past, Temporal Logic [22] and Metric Temporal [3] have been the two primary languages used by REM tools. Some notable temporal-logic based tools are the Temporal Rover and DBRover [4] and NASA's PaX [17]. The major drawback of these approaches is that the assertions are expressed in textual form, which are hard to create and understand. The engineers need to work with two separate languages and formalisms, with assertions written in temporal logic and system designs in statecharts, The lack of a unified formalism requires users to work with two models, one for verification of the assertions and one for specification of the design, with no guarantee that the correctness of one implies the correctness of the other.

StateRover statecharts provide a coherent uniform formalism for both system designs and requirements assertions. It is easier for system designers to create and understand statechart assertions than text-based temporal assertions because statechart assertions are visual, intuitive, and resemble statechart design

models. For example, statechart assertions are event driven just like statechart models, while temporal logic is purely propositional. Moreover, statechart assertions are Turing equivalent and are therefore significantly more expressive than temporal logic. While many commercial tools (e.g. Statemate, Rhapsody, Rational Rose RealTime) provide the capability to generate target code from statechart designs, StateRover supports the specification of and code generation for non-deterministic statecharts, which is essential for the specification and REM of more complex requirements.

Modeling and simulation plays a vital part in the development of network systems. All prominent simulation tools (e.g., OPNET, OMNeT++, Ptolemy) provide a rich set of reusable models to simulate various communication protocols, network devices and links. The availability of these reusable models provide cost-effective means to study the temporal and performance requirements of complex networked systems via run-time monitoring and modeling and simulation early in the development process. Most of these simulation tools support system models that are made up of agents (i.e. black boxes) communicating via message passing. This paper presents an architecture that wraps the StateRover generated code inside these black boxes (called modules) in the OMNeT++ models. Similar architecture can be developed for the StateRover generated code to work with OPNET and Ptolemy as well.

5.2 Reuse of Statechart Assertions

Reuse is one important reason for separating assertions from the statechart model. Java interfaces are a convenient tool for separating components. The StateRover uses two interfaces for primary-statechart/assertion-statechart separation: ITRPrimary is the interface for the primary statechart whereas ITRAssertion is the interface for the assertion statechart. The interfaces are extendible (being in source code form). This allows a primary to pass down custom information to the assertion without the assertion "knowing" who the primary is, and vise versa.

Event mapping is the simple mapping of event names from the primary statechart name space to the assertion name space. While the Assertion2 statechart uses the primary's event name (*heartbeat* in Figure 4b), it could be written with a generic event name such as P. A particular instance of this assertion inside the *LE* primary would then map *heartbeat* to P using the StateRover tool. The StateRover supports name mapping of assertion events to the primary statechart event space and will generate code to implement such mapping.

In [9], we present a process for the development and evaluation of statechart assertions early in the development process. The ability to test the statechart assertions independent of the prototype design ensures that system designers truly understand the required system behavior without being tainted by any pre-conceived solutions. With the help of StateRover's code generator, we can create a library of executable assertion patterns consisting of generic statechart assertions and the accompanying scenario-based test cases. The use of pre-tested

generic statechart assertions will lessen the development time and improve the quality of the statechart assertions in rapid prototyping.

6 Conclusions

This paper brings together several technologies (statechart assertion formalism, run-time monitoring, discrete event simulations, JUNIT based test methodology) to support the behavior modeling and run-time verification of complex networked system temporal requirements. The novelty of the proposed approach include: (1) writing formal specifications using statechart assertions, (2) JUnit-based simulation and validation of statechart design and assertions, (3) automatic, JUnit-based, white-box testing of statechart prototypes augmented with statechart assertions, and (4) the use of discrete event simulation in tandem with run-time verification for networked system prototypes. In addition, this paper presents a clean-slate 4D architecture to support the complex communication requirements of complex networked systems. To demonstrate the code design described in Section 4.2, we have implemented a simple OMNeT++ model consisting of 4 instances of the Decision Element capsule connected in a unidirectional ring. Our next step is to develop a prototype for the highway automated system shown in Figure 11, and test the timing behavior of the design with the different platoon maneuver scenarios.

The StateRover Eclipse plugin is currently under development. This plugin will enable the development as UML statechart models and diagrams under the Eclipse IDE like any C++ or Java file. The StateRover will include support for system-level UML modeling and specification. For the system-level modeling view it will support component diagrams whereas the system-level verification view will support a formal specification language based on message sequence charts. The StateRover will include code generation and REM for both types of diagrams.

Acknowledgments. The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

References

1. Beck, K., Gamma, E.: Test infected: Programmers love writing tests. Java Report. **3(7)** (1998) 37–50
2. Caffall, D., Cook, T., Drusinsky, D., Michael, J.B., Shing, M., Sklavounos, N.: Formal Specification and Run-time Monitoring within the Ballistic Missile Defense Project. Tech. Report NPS-CS-05-007. Naval Postgraduate School, Monterey, California (June 2005)

3. Chang, E., Pnueli, A., Manna, Z.: Compositional Verification of Real-Time Systems. Proc. 9th IEEE Symp. On Logic In Computer Science. (1994) 458–465
4. Drusinsky, D.: The Temporal Rover and ATG Rover. Lecture Notes in Computer Science, Vol. 1885 (Proc. Spin2000 Workshop), Springer-Verlag, Berlin (2000) 323–329
5. Drusinsky, D.: Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions. Proc. 4th Runtime Verification Workshop (RV'04), Invited paper (2004)
6. Drusinsky, D.: Modeling and Verification Using UML Statecharts A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking. ISBN 0-7506-7949-2. Elsevier (2006)
7. Drusinsky, D., Harel, D.: On the power of bounded concurrency I: Finite Automata. J. ACM. **41(3)** (1994) 517–539
8. Drusinsky, D., Shing, M.: Verification of Timing Properties in Rapid System Prototyping. Proc. 14th IEEE International Workshop in Rapid Systems Prototyping. San Diego, California (June 9-11, 2003) 47–53
9. Drusinsky, D., Shing, M., Demir, K.: Creation and Validation of Embedded Assertion Statecharts. Proc. 17th IEEE International Workshop in Rapid Systems Prototyping. Chania, Greece (June 14-16, 2006) 17–23
10. Drusinsky, D., Watney, G.: Applying run-time monitoring to the Deep-Impact Fault Protection Engine. Proc. 28th NASA Goddard Software Engineering Workshop (Dec. 2003) 127–133
11. Easterbrook, S., Lutz, R., Covington, R., Kely, J., Ampo, Y., Hamilton, D.: Experiences using lightweight formal methods for requirements modeling. IEEE Trans. Software Engineering. **24(1)** (Jan 1998) 4–11
12. Graw, G., Herrmann, P., Krumm, H.: Verification of UML-Based Real-Time System Design by Means of cTLA. Proc. 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000) (15-17 March 2000) 86–95
13. Greenberg, A., Hjalmtysson, G., Maltz, D., Myers, A., Rexford, J., Xie, G., Yan, H., Zhang, H.: A Clean Slate 4D Approach to Network Control and Management. ACM SIGCOMM Computer Communication Review. **35(5)** (2005) 41–54
14. Guldner, J., Patwardhan, S., Tan, H.S., Zhang, W.B.: Coding of Magnetic Markers for Demonstration of Automated Highway Systems. Preprints of the Transportation Research Board Annual Meeting, Washington, DC (1997)
15. Harel, D.: A Visual Formalism for Complex Systems. Science of Computer Programming. **8** (1987) 231–274
16. Havelund, K., Pressburger, T.: Model Checking Java Programs Using Java PathFinder. International Journal on Software Tools for Technology Transfer. **2(4)** (April 2000)
17. Havelund, K., Rosu, G.: An Overview of the Runtime Verification Tool Java PathExplorer. Formal Methods in System Design. **24(2)** (March 2004) 189–215
18. JUnit.org, http://www.junit.org/
19. Lüth, L., Niehaus, J., Peikenkamp, T.: HW/SW Co-synthesis using Statecharts and Symbolic Timing Diagrams. Proc. 9th International Workshop on Rapid System Prototyping. (3-5 June 1998) 212–217
20. OMNeT++ Discrete Event Simulation System. http://www.omnetpp.org/
21. Park, M.H., Bang, K.S., Choi, J.Y., Kang, I.: Equivalence Checking of Two Statechart Specifications. Proc. 11the International Workshop on Rapid System Prototyping (21-23 June 2000) 46–51

22. Pnueli, A.: The Temporal Logic of Programs. Proc.18th IEEE Symp. on Foundations of Computer Science (1977) 46–57
23. Selic, B., Gullekson, G. and Ward, P.: Real-Time Object Oriented modeling. John Wiley & Sons (1994)
24. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Unpublished white paper, Rational Software (Apr. 4, 1998) http://www.rational.com/media/whitepapers/umlrt.pdf
25. Tsugawa, S., Kato, S., Tokuda, K., Matsui, T., Fujii, H.: A cooperative driving system with automated vehicles and intervehicle communications in demo 2000. Proc. IEEE Intelligent Transportation Systems Conference (2001) 918–923
26. Tan, H.S., Rajamani, R., Zhang, W.B.: Demonstration of an Automated Highway Platoon System. Proc. American Control Conference. Philadelphia, Pennsylvania (June 1998) 1823–1827.
27. Xu, Q., Hedrick K., Sengupta R., VanderWerf J.: Effects of vehicle-vehicle/roadside-vehicle communication on adaptive cruise controlled highway systems. Proc. 56th IEEE Vehicular Technology Conference. **2** (2002) 1249-1253